Performance of Static and Adaptive Service Discovery Architectures in Response to Node Failures

Christopher Dabrowski
National Institute of Standards and Technology
Gaithersburg, MD USA
1-301-975-3249
cdabrowski@nist.gov

Stephen Quirolgico
National Institute of Standards and Technology
Gaithersburg, MD USA
1-301-975-8426
steveq@nist.gov

Kevin Mills
National Institute of Standards and Technology
Gaithersburg, MD USA
1-301-975-3618
kmills@nist.gov

ABSTRACT
Future service-oriented computing systems will include technology to discover and compose component services, and to detect and adapt to failures. Already industry has developed some competing service discovery architectures and protocols to provide such capabilities. In this paper, we compare performance of three such architectures (static two- and three-party and adaptive two-/three-party) when subjected to node failures. We use simulation to instantiate each architecture with behaviors adapted from known service discovery protocols. We quantify the functional effectiveness achieved for each instantiation under an increasing rate of failures. We then decompose non-functional periods into failure-detection latency and failure-recovery latency. Our results suggest an adaptive architecture yields robustness superior to a static three-party architecture and equivalent to, or slightly better than, a static two-party architecture. While our results find that an adaptive architecture entails higher overhead, we argue that it should prove possible to achieve efficiency similar to a static three-party architecture.

Categories and Subject Descriptors
D.2.11 [**Software Architectures**]: Dynamism and Performance

General Terms
Performance, Design, Reliability, Experimentation

Keywords
Service discovery and composition, failure detection and recovery

1. INTRODUCTION

Future service-oriented computing systems will require technology to discover and compose component services, and to detect and adapt to failures. Already industry has developed several competing service discovery architectures and protocols [e.g., 1-7] that include mechanisms service-oriented systems can use to detect failures, and then to recover by finding and composing remote components. In past work, we characterized the robustness of selected service discovery architectures [8-12], and devised and evaluated some adaptive algorithms to improve performance [13-17]. In one previous paper [12], we investigated the effectiveness and efficiency of selected discovery systems in assisting service-oriented applications to detect failures in remote services and to locate replacements. In that work, we modeled selected discovery strategies and failure-recovery techniques in combination with two major architectures that underlie present-day service discovery systems: two-party, where clients and services rendezvous directly, and three-party, where clients and services rendezvous through a directory. For the two-party architecture, we modeled behaviors adapted from Universal Plug-and-Play (UPnP) [1]; for the three-party architecture, we modeled behaviors from Jini™ Networking Technology (Jini) [2]. While we found the two-party architecture more effective and efficient (than the three-party architecture) in detecting and recovering from node failures, we also noted that the three-party architecture provides better support for large-scale, service-oriented systems, such as grid computing systems [18]. Based on these previous findings, we suspect that some form of adaptive architecture might yield the scalability of the three-party architecture (when directories are available) and the reliability of the two-party architecture (when directories are unavailable).

In this study, we expand our investigation of service discovery systems to include an adaptive (two-/three-party) architecture, which operates as a three-party architecture under normal conditions but which adapts to a two-party architecture when directories are lost to failure. For the adaptive architecture, we model behaviors from the Service Location Protocol (SLP) [3]. Since discovery systems provide middleware to support service-oriented applications, we layer a model of the same application on top of all three architectures (and their underlying behaviors). The application consists of a client that seeks to discover, compose, and use a set of remote services. We restrict client behavior in a form that allows us to specifically focus on fundamental properties of the underlying service discovery systems. For example, we ensure activation of failure-detection and recovery processes by prohibiting clients from caching references to discovered services. As much as possible, we also select client behaviors that expose performance effects arising from architectural differences, while limiting performance effects

Conference'04, Month 1–2, 2004, City, State, Country.

arising from differences in underlying behaviors. To quantify performance differences, we measure the functional effectiveness of the simulated application, that is, the proportion of time that the client can access an operational subset of remote services required to perform the intended use case. To provide a clearer picture of failure response, we decompose functional effectiveness to measure failure-detection latency (i.e., the time required for a client to recognize a remote service has failed) and failure-recovery latency (i.e., the time required for the client to replace a failed service). We also measure overhead as the number of messages generated by the underlying protocol.

The remainder of the paper is organized as four sections. Section 2 introduces the essential concepts underlying service discovery architectures and protocols, including behaviors and parameters (as implemented in our models) associated with discovery, failure detection, and recovery. Section 3 describes the design and parameters used in our simulation experiments. Section 4 presents our simulation results and discusses our fundamental findings and associated causes. We conclude in Section 5.

2. DISCOVERY AND RECOVERY

Service discovery systems enable clients to discover and use remote services over a network. Such systems also include protocols and mechanisms that enable service-oriented components to detect failures, and to respond by restoring connections to remote components or by locating alternate components.

2.1 Service Discovery

A number of different designs have been proposed for service discovery systems. For example, Intel and Microsoft developed UPnP™ to enable plug-and-play by service-oriented components, while Sun Microsystems developed Jini as a general service discovery system to operate atop Java™. In addition, Sun and other companies also created the Service Location Protocol (SLP), which has now been standardized by the Internet Engineering Task Force (IETF). Our analysis of these and other discovery systems revealed that most designs use one of two underlying architectures: two-party or three-party. A two-party architecture consists of two component types: a *service manager* (SM) that provides information on behalf of managed services (which we call *service providers*, or SPs) and a *service user* (SU) that discovers these services on behalf of a client. A SM maintains a set of *service descriptions* (SDs) that associate specific service attributes with a particular SP. The three-party architecture adds a third component type, a *service cache manager* (SCM), which provides an independent directory to hold a set of SDs provided by SMs. The SCM (directory) operates as an intermediary, matching SDs acquired from SMs to queries received from SUs.

The goal of service discovery is to allow a SU (supporting a client) to discover SDs to satisfy specific requirements. In a two-party architecture, service discovery by a SU may take place either passively (i.e., the SU listens for multicast advertisements of SDs by SMs) or actively (i.e., the SU seeks SDs through multicast queries to SMs). In a three-party architecture, both SMs and SUs first seek SCMs either passively (by listening for multicast SCM announcements) or actively (by issuing multicast queries for SCMs). SMs (on behalf of SPs) then register SDs with all discovered SCMs and periodically renew those registrations, while SUs query SCMs for SDs of interest. In this way, SMs and SUs "rendezvous" through SCMs. A three-party architecture can be instantiated with multiple SCMs (i.e., directory replicas) to mitigate the effect of SCM failure and to improve scalability.

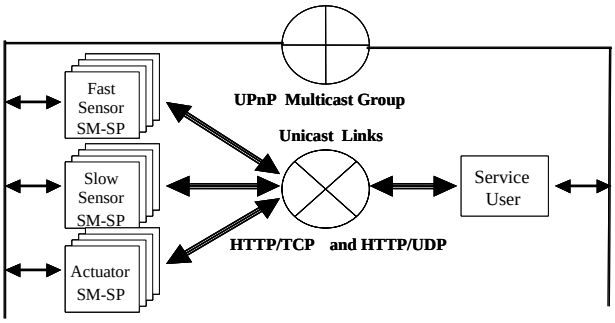


Figure 1. Two-party service discovery architecture with one service user and 12 service manager-service provider pairs

We model service discovery architectures as topologies of various components, where each SM manages (i.e., is paired with) one SP that implements a specific type of service, either a fast sensor, a slow sensor, or an actuator. Our experiment topologies include four SPs of each service type, whose roles are explained below in Section 3. Figure 1 depicts a two-party architecture deployed in a topology of 13 nodes: 12 SM-SP pairs and one SU. In our topologies, each SU, SM-SP pair, and SCM resides on a distinct node. We model the two-party architecture using UPnP behaviors, as described elsewhere [1, 9, 10]. Figure 2 shows the three-party architecture deployed in a topology of 13 to 16 nodes: 12 SM-SP pairs, one SU, and up to three SCMs. We model the three-party architecture using behaviors from the Jini specification, as described elsewhere [2, 9, 10]. Our model of the adaptive (two-/three-party) architecture maintains (when possible) the three-party topology shown in Figure 2, but switches to the two-party topology of Figure 1 when SCMs become unavailable. The adaptive architecture is modeled using behavior from SLP, as described in the relevant specification [3].

2.2 Failure-detection Techniques

To detect failure, components supported by discovery protocols rely on a combination of two techniques: (1) monitoring periodic transmissions from remote components and (2) responding to exceptions signaled (when a message could not be sent successfully) by an underlying (reliable) transport service. Periodic message transmission and monitoring is a key failure-detection strategy used in discovery systems. Components can listen for such recurring messages, much as a heartbeat can be monitored to assess patient health. For example, UPnP requires SMs to periodically announce SD availability, and Jini and SLP require SCMs to periodically announce themselves. Missing an anticipated announcement might indicate failure of a SM or SCM. Similarly, Jini and SLP require SMs to periodically renew service registrations on each SCM. Missing a scheduled renewal indicates to the SCM that the associated SM or SP has failed. At the application level, clients may also maintain regular contact with remote services. For example, a client may receive periodic,

scheduled readings from a remote-sensor service. Missing a scheduled interaction with a remote service might indicate a component failure or a communications blockage.

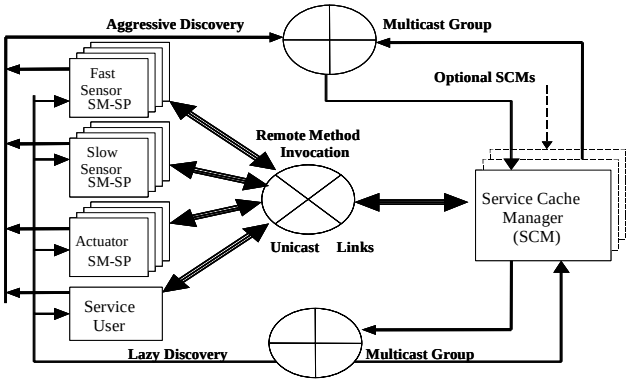


Figure 2. Three-party service discovery architecture with one service user, 12 service manager-service provider pairs, and up to 3 service cache managers

For non-periodic transmissions, components typically use a reliable transport service, which persistently resends unacknowledged messages up to some bound, issuing a remote exception (REX) if that bound is exceeded. For example, if a client attempts to invoke a method offered by a remote service that has failed, then the underlying transport service will eventually return a REX to the client. This outcome occurs in a three-party architecture when a SU attempts to query for a SD from a failed SCM, or when a SM attempts to register a SD with a failed SCM.

2.3 Failure-recovery Techniques

Once a failure is detected, various techniques can be used to recover. Discovery systems generally support two recovery techniques: *soft-state* and *application-level persistence*. The soft-state technique relies on receiving and maintaining transient, or *soft*, information about remote components. Here, remote components issue periodic announcements that convey soft information about the component state, which a receiver can cache for a period of time, consistent with the expected announcement rate. Each announcement may convey updated state information that a receiver can use to overwrite previously cached state information with information from newly arriving announcements. If an announcement fails to arrive, a receiver may discard the previously cached state, effectively eliminating knowledge about the announcing component. If announcements resume, a receiver may rediscover the remote component and recover the latest component state. In our model, the client (and supporting SU) uses a modified form of soft-state recovery, which allows discarded components to be either rediscovered or replaced. For example, in our (static) two-party model, SMs send heartbeat messages to refresh SDs cached by a SU. If the SU fails to receive these heartbeat messages, it will discard the SD as well as knowledge about the related SM.

To effect recovery, SUs in our two-party model commence periodic multicast queries to search for a new instance of a required service. After reacquiring a matching SD, the SU ceases sending queries until the need for a new SD arises again. In our three-party model, loss of contact with a service may cause the SU to query a SCM for a replacement. In addition, a SCM will discard a SD after failing to receive heartbeat messages from SMs to renew the cached SD. In our experiments, SUs (in a three-party topology) poll SCMs at 180-s intervals to check the status of previously discovered SMs. If the SCM response indicates that the previously cached SD has been purged, then the SU assumes the related service has failed. When services (i.e., SM-SP pairs) recover from a failure, discovery procedures (in our models of Jini and SLP behaviors) ensure that the SM rediscovers the SCM within 120 s. SUs can then acquire the corresponding SD by querying the SCM. Of course, SCMs can also fail.

In our model of a static three-party architecture, SCM startup announcements ensure rediscovery of a restarted SCM by SMs and SUs within 30 s after the SCM restarts. If all SCMs have failed, a SU cannot recover a needed SD until at least one SCM recovers. In our model of an adaptive architecture, losing contact with all SCMs causes a SU or SM to issue periodic multicast queries (at 120 s intervals) to search for a SCM. If no SCMs can be found following the first attempt, then the SU changes its strategy and also issues multicast queries for SMs until a needed service is found. The SU continues to interact directly with any discovered SMs while concurrently continuing to search for a SCM. When a SCM is found, direct communication between the SU and SMs ceases and the SU reverts to obtaining services through the newly discovered SCM. In this way, an adaptive architecture may operate with either two or three parties, contingent upon SCM availability.

When failure of a remote component leads to a REX, discovery systems generally expect application-level software to initiate recovery, guided by an application-level persistence policy. This policy may require that a component ignore the REX, retry the operation for some period, or discard knowledge of the (failed) remote component. Since our experiment simulates a real-time control application, our client (and supporting SU) does not persist after a REX, but instead discards knowledge of the failed component; thus, relying on periodic announcements and soft-state techniques to recover from the failure. Once discarded, no interactions occur with a remote component until it is rediscovered. We also employ this policy when SCM failure leads to a REX in response to a SU query or SM registration (or renewal) attempt.

3. EXPERIMENT DESCRIPTION

Previously [12], we conducted a simulation experiment to compare the performance of static two- and three-party architectures, and associated failure-detection and recovery mechanisms. Here, we use the same experiment design, but include a comparison of the static architectures against an adaptive architecture (as described above in Section 2). We simulated the architectures in various topologies, yielding the eight configurations shown in Table 1. Each configuration includes a single client (and supporting SU) and twelve SM-SP pairs (four of each type: slow sensor, fast sensor, and actuator). The configurations differ in terms of the number of SCMs supported.

Each experiment repetition of a particular configuration proceeds along similar lines. The client discovers and activates a set (one

Table 1. Eight configurations of service discovery architecture and topology used in our experiments

Architectural Variant	Protocol Basis	Number of SCMs
Static Two-Party	UPnP	None
Static Three-Party	Jini	One
		Two
		Three
Adaptive Two/Three-Party	SLP	None
		One
		Two
		Three

fast sensor, one slow sensor, and one actuator) of remote services needed to carryout the intended use case. After activation by the client, the fast sensor transmits a reading every 2 s and the slow sensor transmits a reading every 30 s. The client invokes the actuator upon receiving an appropriate combination of readings. We simulate actuation attempts using a uniform distribution with a mean of 60 s. When the client holds one SD for a SP of each type (fast sensor, slow sensor, and actuator) and each of the related SPs is operational, then the application is considered *functional*. If the client lacks SDs for one or more SP type, or if one or more of the SDs held by the client describes a SP that is (unbeknownst to the client) not operational, then the application is considered *non-functional*. We measure accumulated *functional time* over the experiment duration, *D*, during which SM-SP pairs and SCMs (if any) periodically fail and recover. Prior to beginning interval *D*, we run the configuration until discovery completes and the client first becomes functional. To focus exclusively on failure-detection and recovery behavior, we do not permit clients to cache backup services; thus, the client holds at most one SD for each SP type at any time. In the static three-party and adaptive architectures, some additional decisions are necessary. Each SM registers its SD with each discovered SCM, and then renews that registration every 60 s (for slow sensors and actuators) or 300 s (for fast sensors). When a renewal is missed, the SCM purges the associated SD. For each SD discovered through a SCM, the client polls the SCM every 180 s to learn if the SD has been purged. If the SD has been purged, the client assumes that the related SM-SP pair has failed.

3.1 Failure Model

During interval *D*, each node containing a SM-SP pair (or SCM, where applicable) fails randomly and independently, although at least one SM-SP pair of each type always remains active in order to provide the client an opportunity to regain a functional state. Given a failure rate R (with R increasing from 0.1 to 0.9 in 0.1 increments) we can calculate a derived mean time-to-failure $\mu = (1-R)\cdot D$. We randomly select node-failure times from a "stepped" normal distribution with three steps: a 0.15 probability that a failure occurs before time $t = (\mu - 0.2\cdot\mu)$, a 0.7 probability that a failure occurs in the range $t = [\mu - 0.2\cdot\mu, \mu + 0.2\cdot\mu]$, and a 0.15 probability that a failure occurs in a range $t = [\mu + 0.2\cdot\mu, 2\cdot\mu]$. The time of failure is distributed uniformly within each step.

When a node fails, the affected SM-SP or SCM becomes unavailable for a time, chosen from one of three failure classes, each having a different probability *P* of occurrence and an associated duration. Short failures occur with $P = 0.1$ for a fixed period of 135 s; intermediate failures occur with $P = 0.7$ for duration selected uniformly on the interval [180 s, 300 s]; long failures occur with $P = 0.2$ for duration selected uniformly on the interval [480 s, 600 s].

3.2 Metrics

We refer to the time during which a client is in a functional state (i.e., has access to the necessary minimum set of operational SM-SP pairs) as *functional time*. We define *non-functional time*, $\overline{F}$, as the accumulated time over D during which a client is in a non-functional state. We let $F = (D - \overline{F})/D$ denote a client's *functional effectiveness*.

A client may incur a delay before detecting entry to a non-functional state. We refer to this delay as *failure-detection latency*. After detecting a non-functional state, the client may incur additional delay while restoring any lost SM-SP pairs. We refer to this delay as *failure-recovery latency*. Periods of failure-detection latency and failure-recovery latency can overlap when a client loses more than one SM-SP pair.

We properly account for such overlapping periods in $\overline{F}$. We define two logical conditions, one of which can be used to measure failure-detection latency and one of which can be used to measure failure-recovery latency.

1. *Service discard condition*: This condition states that each SD held by a SU must match a SD managed by a SM. Violation of this condition occurs when the SM fails but the SU still holds a SD provided by the SM (i.e., the information about the service is inconsistent between the SU and the defunct SM). When a violation of this condition occurs (i.e., the SM fails), failure-detection latency is accumulated up to the time the SU detects that the SD has failed.

2. *Service discovery condition*: This condition states that a SD managed by a SM should be known to a SU if the SU and SM can communicate over the network, and the SD matches the requirements of the SU. Violation of this condition occurs if the SU detects that the SM has failed (thus the SU knows that the information about the service is inconsistent). When a violation of this condition occurs (i.e., the SU detects the failed SM), failure-recovery latency is accumulated up to the time the SU acquires a new SD matching its needs.

4. RESULTS AND DISCUSSION

For each of the eight configurations in Table 1, we assigned *D* = 1800 s and executed multiple (typically 60) independent experiment repetitions for each value of *R*. We conducted one set of experiments (see Sections 4.1 and 4.2) where at least one SM-SP pair remains available for each service type (so the client always has the possibility to recover a set of necessary services). To confirm our findings under different assumptions, we then ran a second variant of the experiment (see Section 4.4) where all SM-SP pairs may fail (so there can be times when the client has no possibility to recover a set of necessary services). For each

repetition, we recorded functional effectiveness (and its components: failure-detection latency and failure-recovery latency) and the total number of messages exchanged.

4.1 Functional Effectiveness

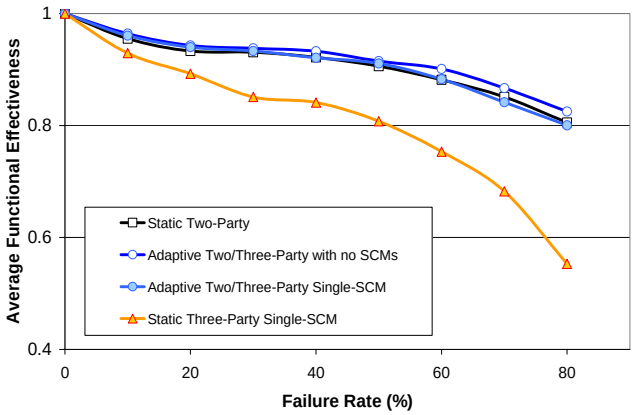
Figure 3 shows the average functional effectiveness of the static two-party, static three-party, and adaptive architectures as *R* increases, but where one SM-SP pair for each service type is always available (implying that the system could be functional for all of *D*). In examining Figures 3a-3c, recall that failure detection occurs when the client misses an expected sensor reading or receives a REX in response to an attempted actuation. In the three-party architecture (whether static or adaptive), failure detection may also occur when, upon polling available SCMs, the client cannot find SDs matching the three SM-SP pairs currently in use. To become functional again, the client must invoke recovery mechanisms to regain or replace failed SM-SP pairs. In the static three-party architecture, at least one SCM must be operational for recovery to succeed. During periods when all SCMs fail, the client is unable to recover needed services, increasing non-functional time. In the adaptive architecture, the client may switch to a two-party operation, searching for SM-SP pairs directly.

Overall, the adaptive and static two-party architectures exhibited high effectiveness, allowing the client to remain functional for as much as 80% of *D* even as the failure rate reaches 80% ($\mu = 360$ s). The mean effectiveness of the static two-party architecture across all failure rates was 0.901. The adaptive architecture showed comparable or better effectiveness regardless of the number of SCMs, achieving mean effectiveness of 0.909 with no SCMs, 0.899 with one SCM, 0.906 with two SCMs, and 0.913 with three SCMs. The static three-party architecture increased in effectiveness with the number of SCMs, reaching (Figure 3c) a rough parity (mean effectiveness of 0.907) with the other architectures as the number of SCMs reached three. Adding SCMs improved effectiveness by lowering the incidence of concurrent failure of all SCMs.

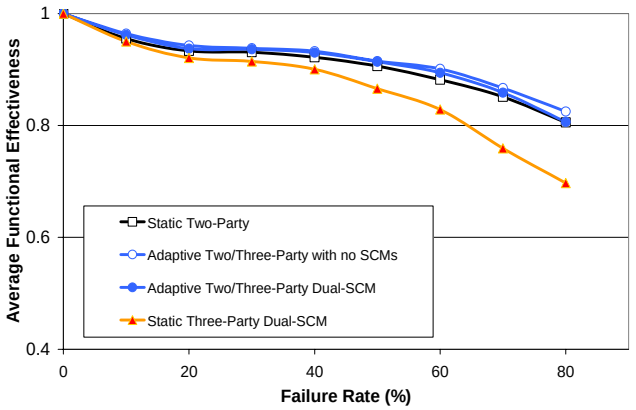
The adaptive architecture required only one or two SCMs to obtain effectiveness comparable to the static three-party architecture with three SCMs. The adaptive architecture without SCMs proved slightly more effective than the static two-party architecture (mean effectiveness of 0.901). We attribute this small difference to the underlying UPnP behavior (static two-party architecture), which requires successful exchange of more messages than the equivalent SLP behavior (adaptive architecture) to transfer a SD from a SM to the client (more details are provided elsewhere [3,9,10]).

4.2 Efficiency

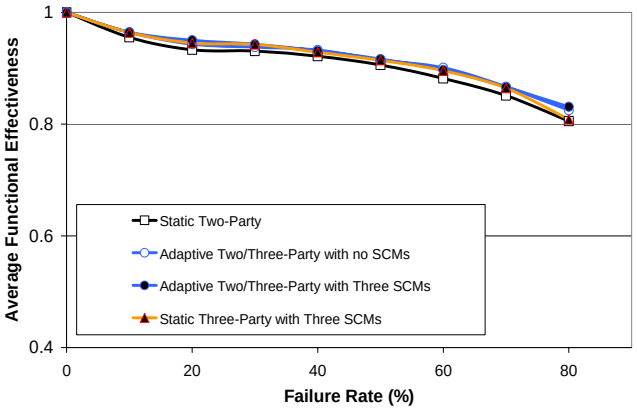
In our simulations, message counts (Figure 4) revealed the static two-party architecture to be significantly more efficient than any configuration of the other architectures, because the two-party configuration does not multicast queries to discover SCMs. The adaptive architecture attempts to discover SCMs even when they do not exist; thus, more messages were generated. When attempting to discover SCMs, the SLP procedures (adaptive architecture) required more messages per discovery attempt than



(a) Static Two-Party, Adaptive Two/Three-Party with Zero & One SCM, and Static Three-Party with one SCM.



(b) Static Two-Party, Adaptive Two/Three-Party with Zero & two SCMs, and Static Three-Party with two SCMs.



(c) Static Two-Party, Adaptive Two/Three-Party with Zero & three SCMs, and Static Three-Party with three SCMs.

Figure 3. Functional effectiveness under increasing *R* where at least one SM-SP of each type is operational (60 reps/point)

the Jini procedures (three-party architecture); thus, for a given number of SCMs one might expect our model of the adaptive architecture to require more messages than our equivalent model of a three-party architecture. This held true for one- and two-SCM configurations; however, as the number of SCMs increased, Jini discovery procedures tended to generate more messages because (in Jini) SCMs attempt to discover each other, while in SLP SCMs do not. This leads us to believe that the adaptive and static three-party architectures would provide comparable efficiencies when deployed with the same number of SCMs, provided that both architectures adopt identical underlying behaviors for discovery, registration, SD renewal, and update propagation.

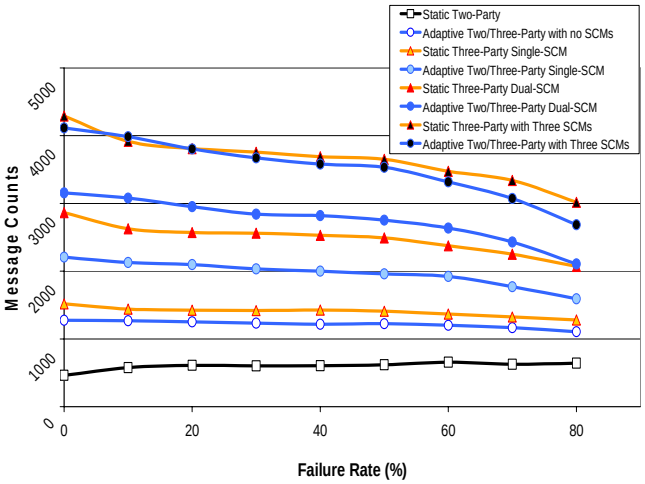


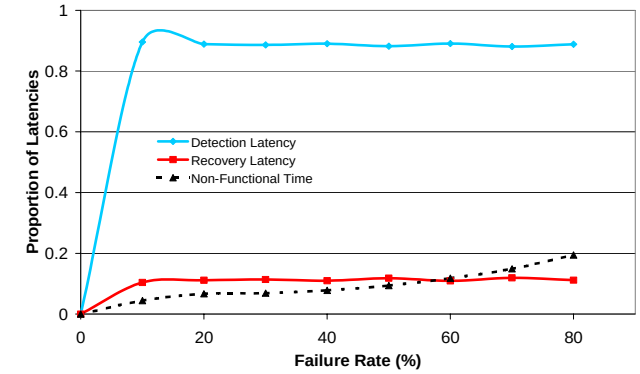
Figure 4. Average message counts under increasing *R* where at least one SM-SP of each type is operational (60 reps/point)

Figure 4 also reveals that, for each configuration with SCMs, total message count decreases as failure rate increases (that is, those curves exhibit a negative slope). This occurs because, as SCMs fail more frequently, the number of SD renewal messages and SCM heartbeat messages decreases. Further, the rate of decline in message counts is inversely related to the number of SCMs employed, that is, the more SCMs in a configuration, the steeper the negative slope. Similarly, SM-SP pairs also fail more frequently, causing the number of queries for SCMs to decline.

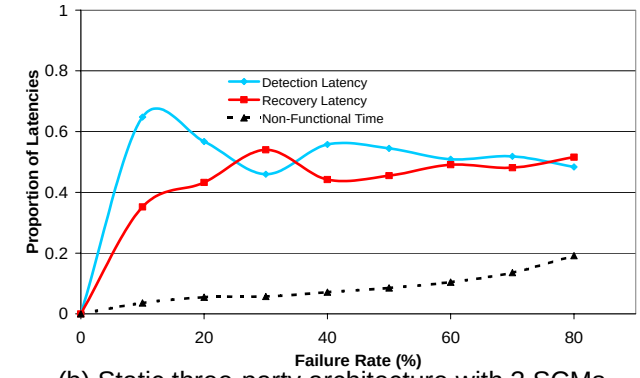
In contrast, the static two-party architecture exhibits an increase in the message count with an increasing failure rate. The increase in message count occurred because, as the failure rate increased, the client more frequently invoked recovery procedures after detecting failed SM-SP pairs. Compare this with the result obtained when the adaptive architecture is deployed without SCMs. In this latter case, message count decreased with increasing failure rate because SLP procedures still require SM-SP pairs to query for SCMs.

4.3 Underlying Causes

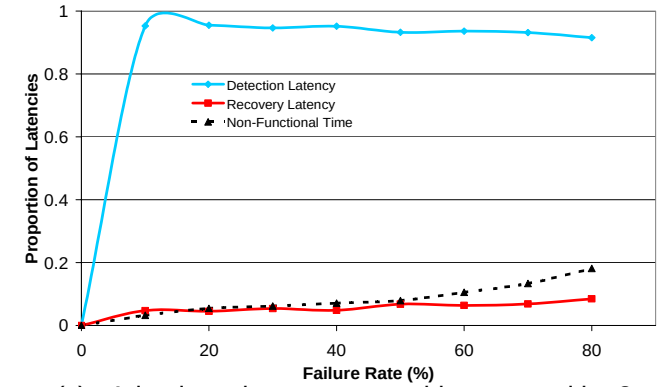
To better understand differences in effectiveness among the alternate architectures, we decomposed (Figures 5a-5c) non-functional time to show the estimated proportion attributable to failure-detection latency and to failure-recovery latency. Figure 5a shows that detection latency is the dominant (~90%) component of non-functional time for the static two-party model.



(a) Static two-party architecture



(b) Static three-party architecture with 3 SCMs



(c) Adaptive three-party architecture with 3 SCMs

Figure 5. Detection and recovery latencies of various architectures and topologies as a proportion of non-functional time (60 reps/point)

Analysis of execution traces showed most failures were detected through missed sensor readings (2 s for fast sensors and 30 s for slow sensors) or REXs received in response to failed actuation messages. Previously [12], we found we could reduce failure-detection latency (and therefore non-functional time) associated with violation of the service discard condition by increasing the registration-renewal frequency for SMs (decreasing the interval between heartbeats).

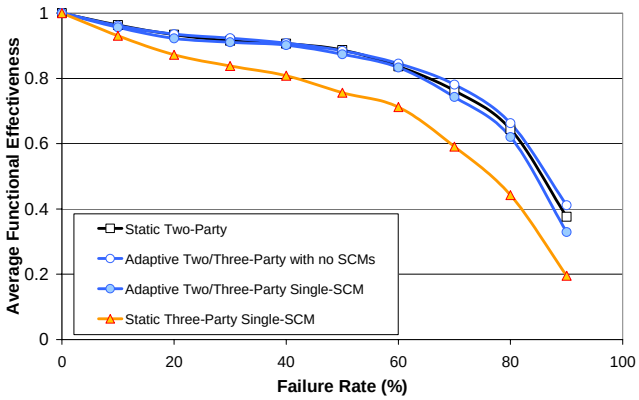
For the static three-party architecture, our data showed that the incidence of concurrent failure of all SCMs increased steadily with increasing failure rate, leading to correspondingly longer periods of time during which the client is unable to find enough SM-SP pairs (and thus remained in violation of the service discovery condition). For the client to regain a functional state, some SCM must first recover, accept SD registrations from available SMs, and respond to queries for SDs from the client. Lacking an ability to directly discover SMs, the client in the static three-party architecture remains non-functional while awaiting recovery of at least one SCM.

These effects are evident in Figure 5b, which shows the proportion of recovery latency increased for the static three-party architecture (with three SCMs) as the failure rate rose. This trend is more marked as the number of SCMs decreases (not shown).

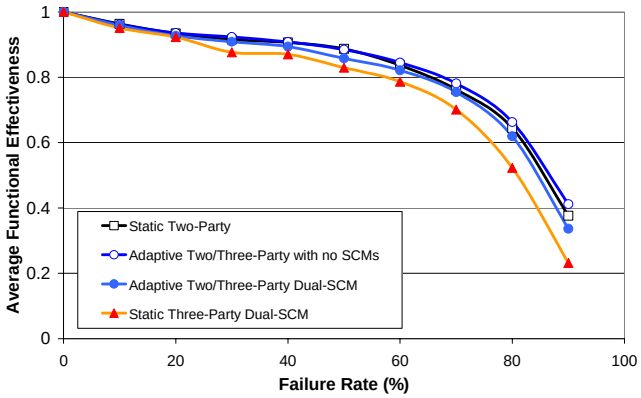
Finally, Figure 5c compares failure-detection latency to failure-recovery latency for the adaptive architecture with three SCMs. Here, the allocation of delay approximates that for the static two-party architecture (see Figure 5a). After detecting a failure and finding no available SCMs, the adaptive architecture adopts two-party discovery procedures, which reduced the proportion of time spent on failure-recovery latency; thus, failure-detection latency again became the greater part of non-functional time.

4.4 Results for Experiment Variant

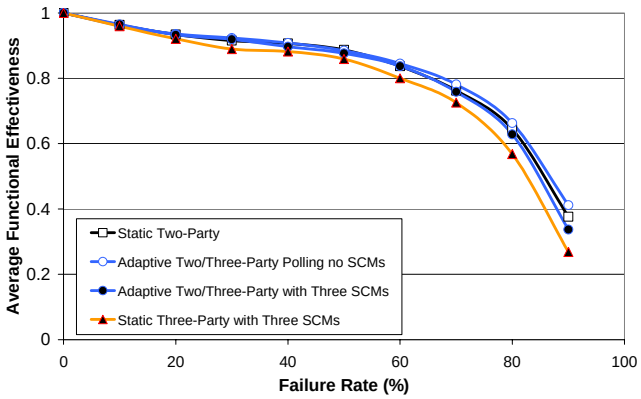
To confirm our findings, we varied the experiment to permit all SM-SP pairs to fail, rather than to have at least one pair always available for each service type. Figure 6 shows these results for the same configurations as in Figure 3. Figures 6a-6c illustrate that functional effectiveness for all architectural variants decreased substantially above R = 60%, as the incidence of concurrent SM-SP failures increased, resulting in extended periods with no SM-SP pairs available for at least one of the service types needed by the client. The curves in Figure 6 show that overall functional effectiveness decreased (from Figure 3) for all configurations due to increased non-functional time arising from situations where all SM-SP pairs of a particular service type became temporarily unavailable. Nevertheless, the comparative ranking of the configurations remained consistent with the results shown in Figure 3. The adaptive architecture with no SCMs again outperformed the static two-party architecture, and the adaptive model also exhibited higher effectiveness than the static three-party model (for the same number of SCMs). Further, the mean effectiveness across all failure rates of the adaptive model with one and two SCMs (0.809 and 0.811 respectively) exceeded the mean effectiveness of the static three-party model with three SCMs (0.787). We note that, in this variant of our experiment, the performance of the static three-party architecture is hindered both by situations where all SCMs are unavailable and where all SM-SP pairs of specific service types are unavailable.



(a) Static Two-Party, Adaptive Two/Three Party with Zero and One SCM, and Static Three-Party with one SCM.



(b) Static Two-Party, Adaptive Two/Three Party with Zero and two SCMs, and Static Three-Party with two SCMs.



(c) Static Two-Party, Adaptive Two/Three Party with Zero and three SCMs, and Static Three-Party with three SCMs.

Figure 6. Functional effectiveness under increasing *R* where all SM-SP pairs of each type are allowed to fail (60 reps/point)

5. CONCLUSIONS

This study compared performance of adaptive and static service discovery architectures under conditions of increasing node failure. The study found that an adaptive architecture exhibits equivalent or better overall functional effectiveness when compared against either a static two-party or three-party

architecture (assuming an equivalent number of directory replicas). These results also suggest that adaptive architectures could achieve a level of efficiency comparable to an equivalent static architecture, when configured with the same topology and implementing the same behaviors for discovery, registration, and renewal. Further, the results found that functional effectiveness of an adaptive architecture configured with one or two directory replicas approaches or exceeds the effectiveness of a static three-party architecture configured with three directory replicas. This means that, while the adaptive and static three-party architectures can both provide independent directories to support service discovery in large-scale systems, the adaptive architecture might yield equivalent or better effectiveness with fewer directory replicas. The adaptive architecture can thus potentially reduce the overhead associated with managing redundant directories. Future work is needed to investigate whether or not an adaptive architecture can successfully employ two-party procedures across a large-scale network.

6. ACKNOWLEDGMENTS

The work described benefits from financial support provided by the National Institute of Standards and Technology (NIST), the Defense Advanced Research Projects Agency (DARPA), and the Advanced Research Development Agency (ARDA). In particular, we acknowledge the support of Susan Zevin from NIST, Doug Maughan and John Salasin from DARPA, and Greg Puffenbarger from ARDA.

7. REFERENCES

[1] *Universal Plug and Play Device Architecture*, V. 1.0, Microsoft, June 8, 2000.

[2] Arnold K., et al. The Jini Specification, V1.0 Addison-Wesley 1999. Latest version is available from Sun.

[3] Guttman, E., Perkins, C., Veizades, J., and Day, M. *Service Location Protocol*, V.2, Internet Engineering Task Force (IETF), RFC 2608, June 1999.

[4] *Salutation Architecture Specification*, V. 2.0c, Salutation Consortium, June 1, 1999.

[5] *Specification of the Home Audio/Video Interoperability (HAVi) Archiecture*, V1.1, HAVi, Inc., May 15, 2001.

[6] *Specification of the Bluetooth System*, Core, Vol. 1, Version 1.1, the Bluetooth SIG, Inc., February 22, 2001, 1999.

[7] *UDDI Version 3.0, Published Specification*, Dated 19 July 2002 (available from http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm#uddiv3)

[8] Dabrowski C. and Mills K., "Analyzing Properties and Behavior of Service Discovery Protocols using an Architecture-based Approach," in the *Proceedings of Working Conference on Complex and Dynamic Systems Architecture*, DARPA-sponsored, December 2001.

[9] Dabrowski C., Mills K., and Elder, J. "Understanding Consistency Maintenance in Service Discovery Architectures during Communication Failure" in the *Proceedings of the 3rd International Workshop on Software Performance*, ACM, July 2002, pp. 168-178.

[10] Dabrowski C., Mills K., and Elder, J. "Understanding Consistency Maintenance in Service Discovery Architectures in Response to Message Loss", in the *Proceedings of the 4th International Workshop on Active Middleware Services*, IEEE Computer Society, July 2002, pp. 51-60.

[11] Dabrowski C. and Mills K., "Understanding Self-healing in Service Discovery Systems" in the *Proceedings of the First ACM SigSoft Workshop on Self-healing Systems* (WOSS '02), November 18-19, 2002, Charleston, South Carolina, ACM Press, pp. 15-20.

[12] Dabrowski, C., Mills, K., and Rukhin, A. "Performance of Service Discovery Architectures In Response to Node Failure," in the *Proceedings of the International Conference on Software Engineering Research and Practice* (SERP'03), CSREA Press June 23-26, 2003, pp. 95-101.

[13] Bowers K., Mills K., and Rose S. "Self-adaptive Leasing for Jini" in the *Proceedings of the IEEE PerCom 2003*, Forth Worth, Texas, March 23-26, 2003, pp. 539-542.

[14] Mills K. and Dabrowski C. "Adaptive Jitter Control for UPnP M-Search", in the *Proceedings of ICC 2003*, May 11-15, 2003 in Anchorage, Alaska.

[15] Rose S., Bowers K., Quirolgico S., and Mills K., "Improving Failure Responsiveness in Jini Leasing", in the *Proceedings of the 3rd DARPA Information Survivability Conference and Exposition* (DISCEX-III 2003), IEEE Computer Society, April, 2003, Vol. II, pp. 103-105.

[16] Bowers K., Mills K., Quirolgico S., and Rose S., "Self-Managed Leasing for Distributed Systems", in the *Proceedings of the 1st Workshop on Algorithms and Architectures for Self-Managing Systems*, co-sponsored by ACM SIGMETRICS, June 2003.

[17] Mills K., Rose S., Quirolgico S., Britton M., and Tan C. "An Autonomic Failure-Detection Algorithm" in the *Proceedings of the 4th International Workshop on Software Performance* (WoSP 2004), January 14-16, 2004, San Francisco, California, ACM Press, p. 79.

[18] Foster, I. et al. *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*. www.globus.org/research/papers/ogsa.pdf, Draft Document, June 22, 2002